# Incremental Compilation for Parallel Logic Verification Systems

Russell Tessier, *Member, IEEE*, and Snigdha Jana

***Abstract*—Although simulation remains an important part of application-specific integrated circuit (ASIC) validation, *hardware-assisted parallel verification* is becoming a larger part of the overall ASIC verification flow. In this paper, we describe and analyze a set of *incremental* compilation steps that can be directly applied to a range of parallel logic verification hardware, including logic emulators. Important aspects of this work include the formulation and analysis of two incremental design mapping steps: the partitioning of newly added design logic onto multiple logic processors and the communication scheduling of newly added design signals between logic processors. To validate our incremental compilation techniques, the developed mapping heuristics have been integrated into the compilation flow for a field-programmable gate-array-based Ikos VirtuaLogic emulator [1]. The modified compiler has been applied to five large benchmark circuits that have been synthesized from register-transfer level and mapped to the emulator. It is shown that our incremental approach reduces verification compile time for modified designs by up to a factor of five versus complete design recompilation for benchmarks of over 100 000 gates. In most cases, verification run-time following incremental compilation of a modified design matches the performance achieved with complete design recompilation.**

***Index Terms*—Incremental compilation, incremental partitioning, incremental routing, logic emulation.**

## I. Introduction

In an effort to provide complete functional coverage, application-specific integrated circuit (ASIC) designers often use hardware-assisted parallel verification platforms. These systems, such as logic emulators [1]–[3] and rapid prototyping systems [4], contain special-purpose logic processors or field-programmable gate arrays (FPGAs), which evaluate logic functions and communicate results in parallel. Although the need for mapping algorithms that can incrementally address design changes is apparent in many areas of computer-aided design (CAD), this need is particularly acute in the area of compilation for hardware-assisted verification. Parallel verification systems often exhibit design-mapping times of hours rather than minutes due to the need for the coordination of system-wide communication and the individual compilation of netlists for numerous logic processors. The integrated nature of the compilation process for these systems limits the capability of designers to incrementally recompile logic designs after small design changes. While these design changes can be integrated into workstation-based design simulation relatively straightforwardly, modified designs targeted to parallel verification generally require long recompilation routines.

The need for incremental design support is a result of recent interest in core-based design and system-on-a-chip integration. Most ASIC verification flows involve numerous iterations of design test, debug, and recompilation. As design modifications are evaluated and design errors are identified, the original design is subjected to a series of minor modifications. Often, design changes are confined to a subset of the original design and affect only a small part of the overall design logic (often less than 15%). For example, a design change may be isolated to a single or small number of register-transfer level (RTL) components that are substantially smaller than the overall design but encompass more logic than will fit in a single logic processor in the verification system. If recompilation for the verification system can be limited primarily to those logic processors that contain logic affected by the design change, the incremental compilation process can be greatly accelerated. The ability to support design changes in this small set of processors is crucial to avoiding the need to recompile all processors in the system from scratch. In addition to providing fast design turnaround, the resulting verification run-time of the incrementally compiled design should be the same as or nearly the same as the verification run-time of the original design mapping.

Existing software systems for parallel verification typically contain a number of automated steps to translate a gate-level or RTL netlist to parallel processing hardware. Included in these steps is a partitioning step to separate the user design into pieces that will fit into each target logic processor, a placement step to select the appropriate processor to hold each design partition, a routing step to interconnect interpartition wires using board wiring resources, and an individual processor compilation step to schedule evaluation of logic expressions. By far the most computationally expensive task of this design flow is individual processor compilation (e.g., FPGA place and route), which can require several hours *per device* given tight timing constraints and logic capacity limitations. Initial compilation of a prototype design may require many hours at the individual processor compilation stage, even if multiple compiles are performed in parallel across a network of workstations.

In this paper, we formulate and develop incremental techniques to identify changed design logic, partition it across affected logic processors (FPGAs) in a parallel verification system, and efficiently determine communication patterns between affected processors. In following this incremental path, attempts are made to limit the number of processors that are affected while minimizing the affect of incremental

Manuscript received June 19, 2001; revised March 3, 2002.

R. Tessier is with the Department of Electrical and Computer Engineering, University of Massachusetts, Amherst, MA 01003 USA (e-mail: tessier@ecs.umass.edu).

S. Jana is with Intel Corporation, Hillsboro, OR 97124 USA.

Digital Object Identifier 10.1109/TVLSI.2002.801614

compilation on overall verification run-time. In an initial step, design changes between the initial and modified design netlist are noted and affected logic processors are determined. These changes include design logic that is *removed* from the original netlist as a result of design modification and new logic that is *added* to the original design. After removed logic is extracted from the original design mapping, added design logic is partitioned across a minimum number of logic processors using a constrained partitioning algorithm. Partitioning is followed by the scheduling of both intraprocessor computation and interprocessor communication for added design logic. These scheduled tasks are coordinated with logic evaluation and communication for logic remaining from the original design. As a final step, a minimum number of logic processors are recompiled to finalize a complete parallel verification image in a fraction of the time of complete system recompilation.

To prove the benefit of the incremental design approaches, we have targeted incremental compilation techniques to an FPGA-based logic emulation system available from Ikos Systems [1]. New partitioning and routing approaches are included in the design flow to augment initial from-scratch design mapping. Five large RTL benchmarks are used to illustrate the compile-time improvement for our approaches. The initial designs have been substantially modified at the RTL level (approximately 10–15% of each netlist was changed) to explore a range of feasible implementation choices. Following design mapping to a commercial emulator, the effect of interprocessor topology on incremental verification efficiency is evaluated.

The rest of this paper is structured as follows. Section II discusses previous work in parallel verification and basic parallel verification design flows. In Section III, the software flow required to support incremental compilation is reviewed. Our incremental approach to design mapping is discussed in Section IV. Section V discusses our experimental methodology. Experimental results that have been derived from our system are described in Section VI. Finally, in Section VII, we summarize this paper and make concluding remarks.

## II. Background

### A. Parallel Logic Verification

Numerous parallel verification systems have been developed to support ASIC designs containing millions of logic gates. These systems generally contain a collection of processing elements, such as logic processors or FPGAs, interconnected in a fixed topology. Although early verification systems contained a small number of processors [5], contemporary architectures [1] contain hundreds of interconnected processing elements, typically enclosed in specially designed cardcages. These systems can obtain prototyping speeds that range from a few megahertz to over 20–30 MHz [6]. Verification performance is usually limited by the physical distribution of emulation memory devices, which are used in verification systems to verify on-chip memory resources, and the need to transport intermediate data values between logic processors. Over the past decade, the verification capacity of parallel verification systems has kept pace with ASIC design sizes through the phenomenon of Moore's law. As design sizes have increased in capacity and performance, so have the processor and memory components that constitute verification systems.

Unlike traditional parallel processing, which often requires time-varying computation and interprocessor communication patterns, the structure of circuits under test is predictable and static. As a result, both processor computation and interprocessor communication can be completely determined at *compile time* through intraprocessor computation and interprocessor communication scheduling. This compile-time approach to mapping has been demonstrated by a sizable collection of contemporary logic emulators. The Quickturn Cobalt verification system [7] contains up to eight boards with 64 processors per board. Due to direct connections between processors, it is possible for each processor to broadcast results to other processors at time slots determined during compilation. The Arkos emulation system [8] contains 14 logic ASICs per board, each containing 32 processing elements. Logic evaluation inside the ASICs and communication between the ASICs takes place at compile-time determined time intervals. A similar system that includes multiple logic processors interconnected across multiple boards is the Tharas Hammer system [9]. Logic processors in this system have been specially designed to allow for rapid, parallel evaluation of logic expressions. For several systems, off-the-shelf FPGAs, rather than custom logic processors, are used as processing elements. The Ikos VirtuaLogic system [1] contains 64 FPGAs per board. These FPGAs are interconnected in a direct-connect network. A statically scheduled interconnect multiplexing technique is used to exchange data between the FPGAs. A similar technique for data communication is employed by the Xtreme emulation system [2] by Axis Systems. This system implements data computation and communication using precompiled single-instruction multiple-data communication patterns. Both inter-FPGA and interboard communication are supported.

### B. Parallel Verification Software Flow

Although parallel verification systems vary considerably in terms of hardware architecture, many similarities exist in their basic design mapping flows. Contemporary compilation software for parallel verification systems has evolved significantly as component logic processor capacities and system sizes have increased. A typical parallel verification system software flow for converting a structural or RTL netlist to a physical realization appears in Fig. 1. *Design translation* converts the original design netlist into the native technology of the verification system [5]. Examples of design translation include RTL synthesis, technology mapping for FPGA-based logic emulators, and Boolean reduction for time-sequenced logic processors. Following design translation, design logic is *partitioned* into pieces that will fit within the logic and memory resources of the logic processors and assigned to specific system resources via *global placement*. Communication between logic processors is determined through two *global routing* steps, which are shaded in Fig. 1. First, feasible shortest paths between logic processors are determined using available board routing resources. Subsequently, communication between processors is scheduled based on logic dependencies and topology constraints. As a final step, *processor compilation* for each logic processor is performed. This

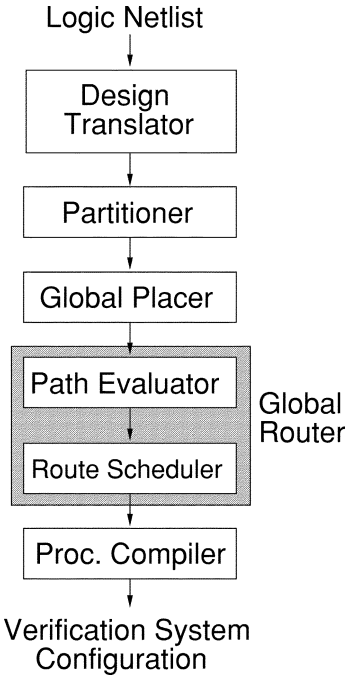


Fig. 1. Parallel verification software flow.

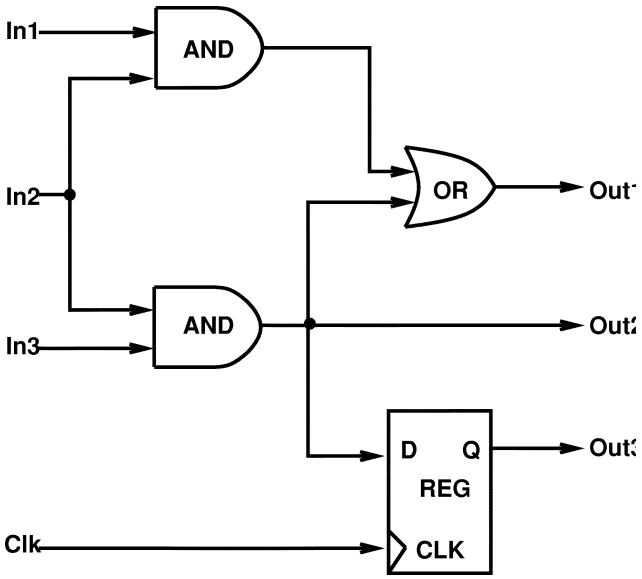


Fig. 2. Dependence calculation example [10]. Dependence is evaluated from inputs to outputs.

step includes FPGA place and route for FPGA-based logic emulators and Boolean instruction scheduling for special-purpose logic processors [8], [9].

### C. Virtual Wires Emulation

The target system for this paper is an Ikos VirtuaLogic emulation system that contains 64 Xilinx XC4036EX FPGAs. Logic evaluation and communication in VirtuaLogic systems is based on Virtual Wires technology [10], a heuristic list scheduling approach. This technique pipelines multiple logical signals called *virtual wires* across single inter-FPGA wires to overcome FPGA pin limitations [10]. The derived communication schedule establishes a feasible space–time route for every logical wire while guaranteeing that all FPGA combinational dependencies are correctly ordered.

The design mapping steps for Virtual Wires systems follow the basic software flow outlined in Fig. 1. In a typical system, RTL synthesis and FPGA technology mapping are performed during the design translation stage. This is followed by design partitioning into logic blocks small enough to fit within FPGAs, placement of logic blocks onto specific FPGAs, and scheduling of both intra-FPGA logic evaluation and inter-FPGA communication. Babb [10] and Hauck and Agarwal [11] provide additional detailed information on Virtual Wires compilation. A distinctive aspect of parallel verification systems in general and Virtual Wires systems in particular is computation and communication scheduling. The initial step in Virtual Wires scheduling is the determination of all circuit combinational dependencies. Logic can be scheduled for evaluation once all dependent inputs have reached a known value. After intermediate values have been determined, the scheduling approach pipelines multiple logical signals (virtual wires) across inter-FPGA wires to overcome FPGA pin limitations. Sequential primitives (flip-flops) and design primary outputs form combinational boundaries for logic evaluation in each design clock cycle. These values are evaluated at the end of each design clock cycle after all combinational dependencies have been satisfied.

As first described by Babb [10], a key aspect of signal scheduling is the determination of signal *depth* and *dependence*. Both depth and dependence apply to interpartition wires. To analyze input to output *dependence,* we determine the set of outputs to which a combinational path exists from each input. An output is said to be a dependent (or a child) of an input if a change in that input can combinationally change the output. In determining dependence, we assume that all outputs of a combinational library primitive are dependents of all the inputs of that primitive. Similarly, no outputs are dependents of any of the inputs for sequential primitives.

Let Depend$[i]$ denote the set of outputs of a given partition that are dependents of an input of the same partition connected to an interpartition wire $i$. Similarly, let $D^{-1}[i]$ represent the set of inputs that are ancestors to an output driving an interpartition wire $i$. By our definition, inputs to storage elements and external outputs will have no dependents—Depend$[i] = \emptyset$—and outputs of storage elements as well as external inputs will have no ancestors—$D^{-1}[i] = \emptyset$.

Fig. 2 shows an example circuit partition containing four interconnected primitive logic elements with three inputs (not including the clock) and three outputs. The dependence relationships for this partition are as follows:

- Depend$[\mathrm{In1}] = \{\mathrm{Out1}\}$;
- Depend$[\mathrm{In2}] = \{\mathrm{Out1}, \mathrm{Out2}\}$;
- Depend$[\mathrm{In3}] = \{\mathrm{Out1}, \mathrm{Out2}\}$.

Likewise, the ancestors are as follows:

- $D^{-1}[\mathrm{Out1}] = \{\mathrm{In1}, \mathrm{In2}, \mathrm{In3}\}$;
- $D^{-1}[\mathrm{Out2}] = \{\mathrm{In2}, \mathrm{In3}\}$;
- $D^{-1}[\mathrm{Out3}] = \emptyset$ (REG is a storage element).

The *depth* calculations use the dependence relationships. The depth of interpartition wire $i$ is the largest number of partitions in a forward combinational path starting at that wire. Depth is

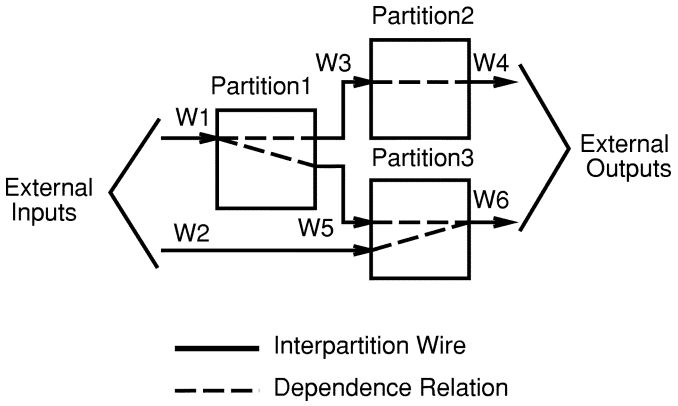


Fig. 3. Depth calculation example. Dependence relations between partition inputs and outputs are represented with dashed lines.

computed recursively from the wire dependence sets such that for each wire $i$

$$\text{Depth}[i] = \begin{cases} 0, & \text{if Depend } [i] = \emptyset \\ 1 + \max_{j \in \text{Depend}[i]} \text{Depth}[j], & \text{otherwise.} \end{cases} \quad (1)$$

Fig. 3 shows an example design with three partitions and six interpartition wires. The dashed lines denote input–output dependence relationships. In this example, wires are at the following depths:

- depth 0: W4, W6;
- depth 1: W2, W3, W5;
- depth 2: W1.

Following dependency analysis, routing in systems using virtual wires starts with the formation of a channel graph [11]. As shown in Fig. 4, this graph is derived from the emulation system topology and each edge represents a unidirectional routing channel. Each channel has a width $C^w$ representing the number of physical wires in the channel. As routing resources in a channel are consumed, the cost of using the channel $C^d$ increases.

Routing for systems using virtual wires is simplified by the abstraction of both time and space. Both logic evaluation and signal communication in Virtual Wires systems are controlled by a high-speed clock called a *virtual clock* [11]. This clock serves as a discrete timebase, providing a reliable mechanism for controlling the order of events at a fine granularity. Since many combinational evaluations and signal transfers may occur in a single design clock cycle, the virtual clock by necessity runs at a much higher frequency than the design clock. Routing for each interpartition wire produces a source–destination *path* in both time and space. Signals that travel a multi-FPGA distance in the system topology are pipelined into a set of intermediate single-FPGA steps. Each signal is registered at FPGA boundaries in a flip-flop synchronous to the virtual clock and is communicated between a pair of FPGAs over a physical wire. As a result, long combinational paths are broken into a series of discrete time steps. Discretization of both time and space results in reliable and predictable verification system operation. For inter-FPGA communication, all partition inputs must be valid before dependent partition outputs are routed to other points in the verification system. All inter-FPGA signals are ordered so that they may be routed as soon as precedence conditions are met. This ordering results in an ordered list of inter-FPGA signals to be routed.

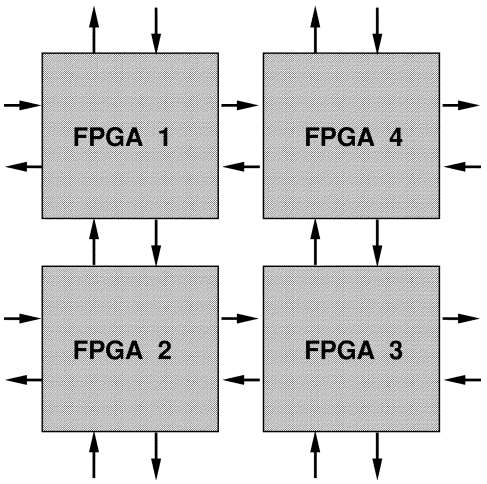


Fig. 4. Near-neighbor FPGA topology. Each arrow represents a unidirectional inter-FPGA channel.

Although a number of routing algorithms exist for time-scheduled verification systems [10], [12], [13], the basic steps required to perform scheduled routing in each are similar. Routing for each individual interpartition signal in Virtual Wires systems involves a series of steps after the signal is selected from the dependence-ordered signal list. The capacity of an inter-FPGA channel $C^w$ can be measured on a time-sliced basis as $C^w(t)$.

To illustrate route timing, routing notation from [11] is used to indicate the fastest possible critical path routing via a series of routing steps.

1) Each signal route starts at a source FPGA $s_f$. The corresponding destination FPGA $d_f$ is the endpoint for the route.
2) The shortest feasible path $P_{sd}$ between partitions $s_f$ and $d_f$ in terms of channels is determined.
3) The *send time* $T_s$ of the signal is determined. This is the time slot at which a signal leaves FPGA $s_f$.
4) The signal arrives at FPGA $d_f$ at the *arrival time* $T_a$ of the signal. The arrival time is defined as

$$T_a = T_s + n \quad (2)$$

where $n$ is the number of FPGA chip boundaries (hops) between source FPGA $s_f$ and destination FPGA $d_f$.

After all dependent signals arrive at an FPGA, the evaluation of combinational logic requires one virtual clock cycle.

The above routing steps are illustrated through the use of an example. The circuit shown in Fig. 5 has been partitioned onto the FPGA topology shown in Fig. 6, which supports near-neighbor communication. Each inter-FPGA signal can only travel between two FPGAs during each system (virtual) clock cycle. In the figure, portions of the original circuit are left unshaded while communication pipeline flip-flops controlled by the virtual clock are darkly shaded. Note that signal $b$ passes unchanged through FPGA 3 on the path from FPGA 2 to FPGA

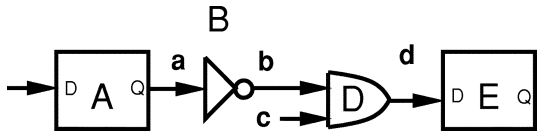


Fig. 5. Input circuit.

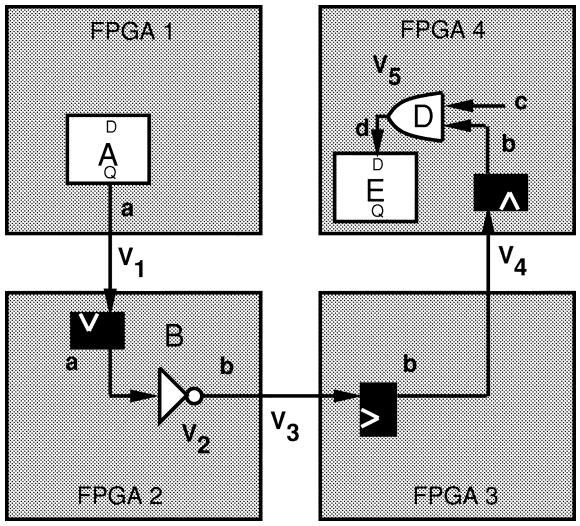


Fig. 6. Circuit mapping to FPGAs for the circuit shown in Fig. 5. Virtual Wires pipeline flip-flops are darkly shaded. Signal **b** is transported from FPGA 2 through the pipeline flip-flop in FPGA 3 to FPGA 4.

4. This *through-hop* is necessary given the lack of a direct FPGA 2 to FPGA 4 connection. Circuit communication in terms of virtual clock cycles can be determined by evaluating the critical path from signal $a$ to signal $d$, as shown in Fig. 7. In Fig. 7, virtual cycles are indicated with labels V1 through V5; communication delays are indicated with notation $n$ equal to a number, where $n$ is the number of virtual cycles required for communication; and combinational evaluation in terms of virtual cycles are indicated with a numeral (e.g., 1). After virtual cycle V5, signal $d$ is latched into flip-flop $E$, completing the design clock cycle.

## III. Incremental Compilation Software Flow

Most design changes that are mapped to parallel verification systems are localized to an isolated region of design logic such as an RTL functional unit or IP core. Since attempts are made to minimize interprocessor communication during initial design mapping, many localized design changes can be isolated to a small number of logic processors and communication paths. Ideally, to support incremental compilation, mapping operations such as partitioning can be isolated to only those logic processors affected by the change and routing can be restricted to a minimum number of routing channels that interconnect the processors. Since individual processor compilation is the dominant component of system compile time, if incremental compilation can be restricted to a minimum number of verification system resources, the number of individual processor compilations can be limited. Typically, when design changes are made to a user design, some existing logic is replaced with new logic. As a result, the size and interconnect structure of the changed piece of logic must fit within the available resources of the system logic

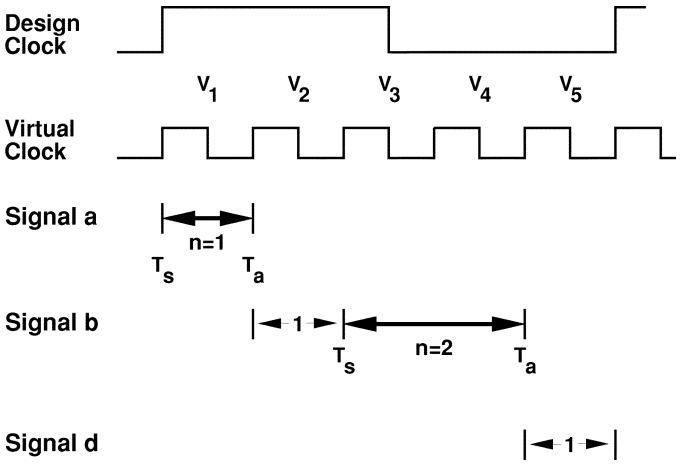


Fig. 7. Design clock cycle for the circuit mapping shown in Fig. 6.

processors. In our new design flow, steps to incrementally alter verification system configuration are developed. These steps include the partitioning of added logic to a subset of system logic processors (FPGAs) and the routing of new design nets across routing resources that are no longer needed by removed logic or that are unused in the initial design mapping. Whenever possible, the verification system configuration is kept unchanged for design logic and signals that have not been modified from the original design.

An incremental compilation flow for parallel verification systems is shown in Fig. 8. These steps can be characterized as follows and will be described in detail in Section IV.

1) *Netlist comparison:* The first step in the incremental compilation process is to identify the disjoint set of logic and interconnect associated with two distinct designs: the *initial design* and the design created after the initial design is modified, the *modified design*. Logic removed from the initial design was assigned to a set of processors as a result of initial design mapping. These *modified processors* provide a possible destination for added logic.
2) *Incremental path identification:* Virtual Wires systems require scheduled data transport between source and destination FPGAs. As shown in Fig. 6, in a system topology with less than direct point-to-point connectivity between all logic processors, individual processors may serve as both processing elements and through-hop steps for intermediate routes. If full connectivity of modified processors using only modified processors cannot be achieved, intermediate *unmodified* processors, that have had no logic removed, may have to be included in incremental routing paths. If used as through-hops, these processors require recompilation to include routing changes. To limit compile time, the number of unmodified processors selected to perform through-hop routing should be minimized.
3) *Incremental partitioning:* Once the modified and required through-hop processors have been identified, newly added design logic can be partitioned onto these processors subject to processor logic and memory capacity constraints.
4) *Incremental routing:* Following incremental partitioning, routing is performed to create a path for the added

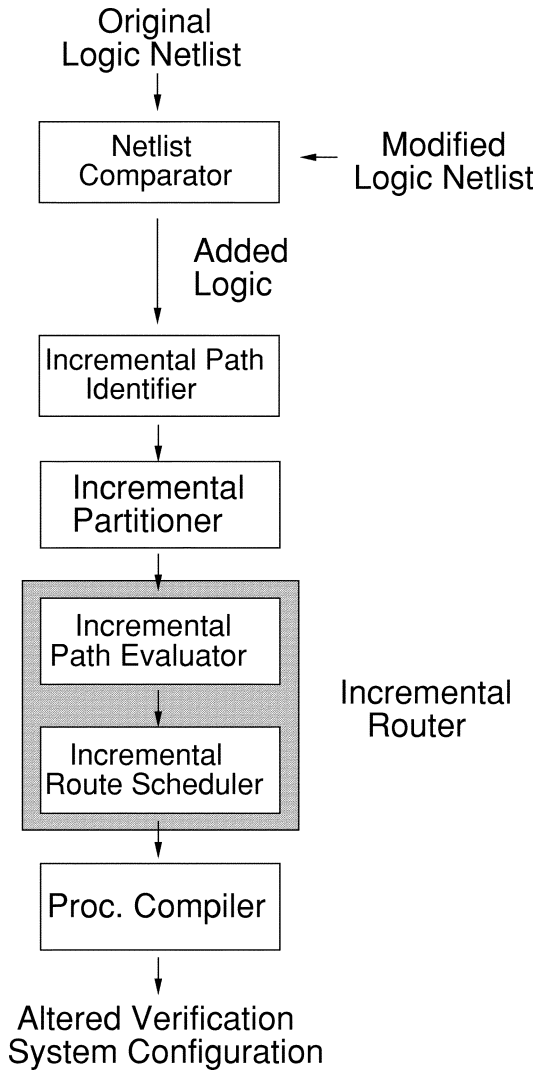


Fig. 8. Incremental compilation software flow.

design signals connecting the modified logic processors. Since other logic processors surrounding the modified logic processors are unaltered, this incremental routing must be performed using board-level routing resources that have not previously been consumed by unchanged design routes. First, feasible shortest paths between logic processors are evaluated. Subsequently, incremental scheduling is used to form a communication pipeline.

5) *Processor compilation:* As a final step, processors that have received new logic or have been modified to include through-hops are recompiled to complete the remapping process.

To date, few practical attempts have been made to integrate incremental CAD algorithms into verification design flows. Although graph algorithms such as bipartitioning and scheduling have matured in recent years, few incremental versions of these algorithms have been developed, and even fewer have been reduced to practice in a working verification system. Cong and Sarrafzadeh [14] propose an incremental partitioning approach based on Kernighan–Lin bipartitioning [15]. In that work, an initially partitioned network is modified and new logic is distributed by an incremental partitioner across partitions that previously have had initial logic removed. The paper indicates that not only should added logic be allowed to move between partitions but also some portion of previously assigned logic should be allowed to move. While the study formulates the incremental partitioning problem, partitioning is not applied in the context of other verification flow steps that can constrain logic placement.

Unlike incremental partitioning, incremental scheduling has been developed previously in several areas of computer-aided design. An incremental scheduling approach for parallel processing [16] allows for run-time dynamic scheduling of interprocessor communication based on computation results. This run-time approach relies on data-dependent computation, which is not necessary for parallel verification of static circuits. DeMicheli [17] and Patasman [18] apply incremental scheduling algorithms to high-level synthesis to refine an initial schedule. The approaches are based on hill-climbing techniques and are appropriate only for limited problem sizes. These iterative improvement algorithms have been extended for low-level retiming [19] and integrated with value prediction. Generally, schedule modifications are localized to a small part of the overall schedule, which make the algorithms difficult to apply to parallel verification. In an earlier version of the work presented in this paper, Tessier presents a greedy incremental scheduler for logic emulation [20]. Perhaps the most relevant incremental scheduling work has been performed in regards to land-based data transmission networks. Ma and Lloyd present an incremental scheduling technique to assign new communication to time-division multiplexed timeslots [21]. An optimal scheduling approach based on linear programming is formulated to allocate additional bandwidth. Varma and Chalasani introduce an incremental graph coloring approach to determine broadcast time slots in an on-line fashion [22]. Neither of these two optimal approaches can scale to support the number of wires typically required in incremental compilation for parallel verification systems.

## IV. Incremental Compilation Algorithms

Our incremental partitioning and incremental scheduling algorithms are similar in nature to their initial-run counterparts but are constrained to minimize perturbation of unchanged logic that has previously been mapped to the verification system. Before describing specific design implementations, each design problem is formulated and problem constraints are noted. Although formulations are made from the perspective of an FPGA-based logic emulation system (the testbed for this work), the approach can be applied to any parallel verification system that follows the software flow shown in Fig. 1.

An initial user design under verification consists of $N$ nodes (gates or flip-flops) and $E$ edges (nets). As a result of the initial mapping flow described previously, a design is mapped to $B$ partitions (FPGAs) in a parallel verification system. A subset $L$ links of $E$ design edges is communicated between FPGA partitions. To create a modified design from an initial design, $N_{\rm sub}$ nodes and $E_{\rm sub}$ edges are subtracted from the initial design and $N_{\rm add}$ nodes and $E_{\rm add}$ edges are added to the design. As a result of this design modification, $B_{\rm sub}$ FPGAs have logic removed and $L_{\rm sub}$ routed links in the verification system are removed.

TABLE I
PARAMETER DEFINITIONS FOR INCREMENTAL COMPILATION

| | **Initial Design Parameters** |
|---|---|
| $N$ | Number of initial logic nodes |
| $E$ | Number of initial design nets |
| $B$ | Number of design partitions |
| $L$ | Number of links to be routed |
| | **Incremental Design Parameters** |
| $N_{sub}$ | No. of nodes subtracted from initial design |
| $E_{sub}$ | No. of design nets subtracted from init. design |
| $B_{sub}$ | No. of design partitions with removed logic |
| $L_{sub}$ | No. of routed links to be removed |
| $N_{add}$ | No. of nodes added to initial design |
| $E_{add}$ | No. of design nets added to initial design |
| $B_{add}$ | No. of target partitions for added logic |
| $L_{add}$ | No. of added links to be routed |

The goal of incremental partitioning is to assign $N_{\text{add}}$ nodes to partitions $B_{\text{add}}$ so that inter-FPGA bandwidth is minimized. A subset $L_{\text{add}}$ of $E_{\text{add}}$ edges must be scheduled for routing to create a completed design. These links can be scheduled in free time slots or in time slots previously used by $L_{\text{sub}}$. The determination of $B_{\text{add}}$ from $B_{\text{sub}}$ is nontrivial and will be described in Section V. Incremental compilation parameters are summarized in Table I.

In some cases, it may be possible to contain all changed logic inside one partition. Examples of these changes include a single gate change or a design flip-flop insertion. For these trivial modifications, there is no need for incremental partitioning or scheduling since only the affected FPGA needs to be recompiled. In this paper, we evaluate design changes that affect multiple FPGAs.

### A. Identification of Changed Logic (Netlist Comparison)

The first algorithmic step in the incremental compilation process, represented by the top box in Fig. 8, is the identification of deleted and added design logic and nets and associated FPGAs affected by the change. This netlist comparison is made in our system through application of depth-first matching applied to both initial and modified design netlists. During the search, the hierarchical names of logic and nets in the modified design are matched against those in the original to identify design changes. It is assumed that any added logic and nets will have a different hierarchical name, which is consistent with the insertion/deletion of an RTL module. Any FPGA that contains logic from the initial design that has been removed requires recompilation to support correct evaluation of the modified circuit. As a result, these modified FPGAs make desirable targets for added design logic. Added logic can be partitioned into FPGAs that have been modified, as space permits.

### B. Incremental Path Identification

Once added nodes $N_{\text{add}}$ have been identified through netlist comparison, but before added nodes $N_{\text{add}}$ can be partitioned onto verification system FPGAs, the specific set of destination FPGAs $B_{\text{add}}$ must be determined. After analysis of the modified and original netlists to determine $N_{\text{sub}}$, the FPGAs with removed logic $B_{\text{sub}}$ can be straightforwardly determined.

Only FPGAs in set $B_{\text{add}}$ are ultimately recompiled. Therefore, for routing links $L_{\text{add}}$, it is necessary to ensure that there is a path from every FPGA in $B_{\text{add}}$ to every other FPGA in $B_{\text{add}}$ that only uses FPGAs in $B_{\text{add}}$, forming a *completely connected* set. Ideally, $B_{\text{add}}$ would be limited to $B_{\text{sub}}$, the FPGAs with removed logic. However, due to topology-related restrictions in the verification system hardware, the FPGAs in $B_{\text{sub}}$ may not form a completely connected set for routing. As a result, FPGA set $B_{\text{sub}}$ may require augmentation with a minimum number of *unmodified* FPGAs to form $B_{\text{add}}$, a completely connected set of FPGAs. Consider, for example, the circuits shown in Fig. 9. It is apparent from the figure that the original inverter circuits on the left have been replaced with a buffer $X$ and an OR gate $Y$. Since FPGA 1 and FPGA 3 contain deleted logic, they form $B_{\text{sub}}$. In the modified circuit, a routing path is required between FPGA 1 and FPGA 3 to complete connectivity. As shown in the figure, FPGAs are interconnected in a near-neighbor topology. Even though FPGA 2 is not in $B_{\text{sub}}$, it must be recompiled to accommodate the transfer of signal $x$ from FPGA 1 to FPGA 3. As a result, $B_{\text{sub}}$ must be augmented to include FPGA 2. Logic previously assigned to these FPGAs that has not been removed remains intact within these partitions.

The FPGAs in $B_{\text{add}}$ are determined through a heuristic evaluation of shortest paths between FPGAs in $B_{\text{sub}}$. A detailed listing of the algorithm is shown in Fig. 10. In an attempt to minimize overall distance, the search algorithm orders all FPGA connections by Manhattan distance. Candidate paths between the FPGAs are then evaluated based on FPGA cost $C(k)$:

$$C(k) = \begin{cases} 0, & k \in B_{\text{sub}} \\ \frac{1}{n}, & k \notin B_{\text{sub}}, \text{ in } n \text{ paths} \\ 1, & k \notin B_{\text{sub}}, \text{ in } \varnothing \text{ paths.} \end{cases} \tag{3}$$

Modified FPGAs in $B_{\text{sub}}$ make the most desirable path candidates since they will be recompiled to include new design logic. Unmodified FPGAs shared by many paths make the second-most desirable candidates since they minimize the total number of affected FPGAs. For each source–destination connection, a cost-based queue of paths $PQ$ is formed to hold candidate path FPGAs. At each search step, the lowest cost FPGA in the shortest path is selected until a path $PT$ is formed. The algorithm includes several search iterations for each path to locate minimum cost paths. If two sequential search iterations result in the same overall cost, the search is terminated. Following the iterations, the unmodified FPGAs located in the search are added to $B_{\text{sub}}$ to form $B_{\text{add}}$.

### C. Incremental Partitioning

In assigning added logic to system FPGAs, the following objectives must be optimized.

1) New logic $N_{\text{add}}$ must be partitioned onto FPGAs $B_{\text{add}}$ such that the sum of the number of links interconnecting $B_{\text{add}}$ is minimized.
2) A number of links from the original design that were routed during initial design mapping may *drive* added design logic. If possible, this logic should be assigned to $B_{\text{add}}$ such that previously scheduled link communication can be used.

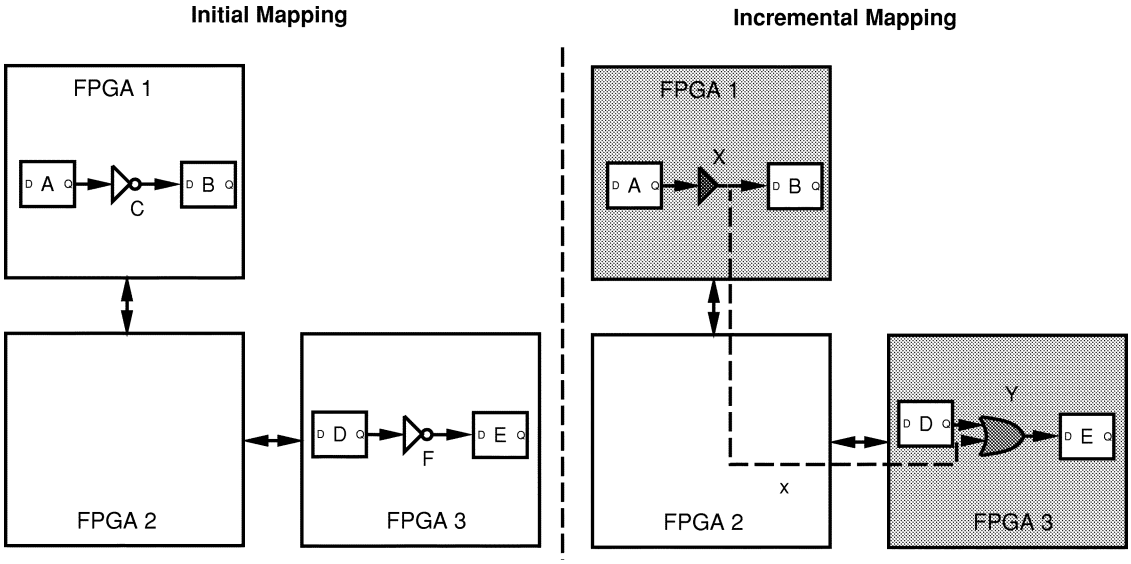


Fig. 9. An example of the need to identify incremental paths. In the subfigure on the right, FPGA 2 must be added to $B_{\text{add}}$ to allow routing from FPGA 1 to FPGA 3.

**i, j:** FPGAs in $B_{sub}$
**PT(i,j):** list of FPGAs in $i$ to $j$ path.
**PQ(i,j):** Candidate FPGAs for inclusion in $PT$.
**C(k):** Cost of FPGA $k$.
**CostT:** $\sum_{all\ FPGAs} C(k)$.

**Order** all $i, j$ connections by Manhattan distance.
**For** each $i, j$
  **Identify** shortest path with Dijkstra's algorithm.
**EndFor**
**Set** $CostT$ to $\infty$.
**While** $CostT$ can be reduced.
  **For** each $i, j$ pair.
    **Rip** up $PT(i,j)$ and update assoc. $C(k)$ values.
    **Initialize** $PQ(i,j)$ with source $i$.
    **While** $j$ not reached
      **Remove** lowest cost FPGA $k$ from $PQ(i,j)$.
      **Add** $k$ to PT(i,j) and update $C(k)$.
      **Update** $CostT$
      **Put** neighbors of $k$ in $PQ$ with cost
        from Eq. (3)
    **Endwhile**
    **Update** FPGA costs in $PT(i,j)$ using Eq. (3).
  **EndFor**
  **Update** $CostT$
**EndWhile**
**Add** selected FPGAs for $i, j$ paths to $B_{add}$.

Fig. 10. Algorithm to identify minimum number of FPGAs for $B_{\text{add}}$. The set of FPGAs with removed logic ($B_{\text{sub}}$) is augmented with the minimum set of FPGAs needed to create completely connected routing paths.

3) A number of links from the original design that were routed during initial design mapping may be *driven* by added design logic. If possible, this logic should be assigned to $B_{\text{add}}$ such that previously scheduled link communication can be used.

These objectives can be met by constraining K-way partitioning across $B_{\text{add}}$ partitions. Before describing the specific steps involved in partitioning, objectives 2) and 3) above are described in greater detail.

*1) Prerouted Input and Output Links:* Often, when logic is replaced in a design, the signals that interface to the logic from the rest of the circuit stay intact. For example, if an RTL core is replaced, the internal structure of the new logic may differ greatly from the old but the hierarchical interface of external signal names and data bus widths remains the same. These interface signals form a special case for incremental mapping. If the interface signals span multi-FPGA boundaries, it may be possible to use routing information from the scheduling of initial-design routing links, limiting the need for routing changes.

This routing optimization can only be performed if the route destination (for driven logic that is added) and route source (for signal sources that are added) are located in the same partitions as original sinks and sources. This placement constraint can be used to guide incremental partitioning. As will be shown in the following example, the use of previously routed links avoids the need to reroute preexisting signals and, in some cases, the need to recompile FPGAs that contain no design changes.

As an example, consider the designs shown in Fig. 11. In the original design shown on the left, two subcircuits have been partitioned across four FPGAs. Communication between FPGAs is pipelined via shaded flip-flops inserted during design routing. On the right, it can be seen that the OR gate $F$ has been replaced with inverter $X$, and flip-flop $B$ and the subcircuit in FPGA 2 have been removed. Inter-FPGA links $a$ and $b$ are present in both designs. During the initial router pass for the original design, the communication links $a$ and $b$ are scheduled. In the modified design, two FPGAs (FPGA 2 and FPGA 4) have had logic removed and require recompilation. This gives two possible partitioning destinations for the inverter and two possible routing paths for $a$ and $b$. These choices are shown with dashed lines in the figure.

If one takes previous routing into account, FPGA 4 is the better target partition for inverter $X$, since previously scheduled $a$ and $b$ communication circuitry in FPGA 1 and FPGA 3 can be used. If FPGA 2 is chosen as the destination for the inverter during incremental partitioning, FPGA 1 will have to be recom-

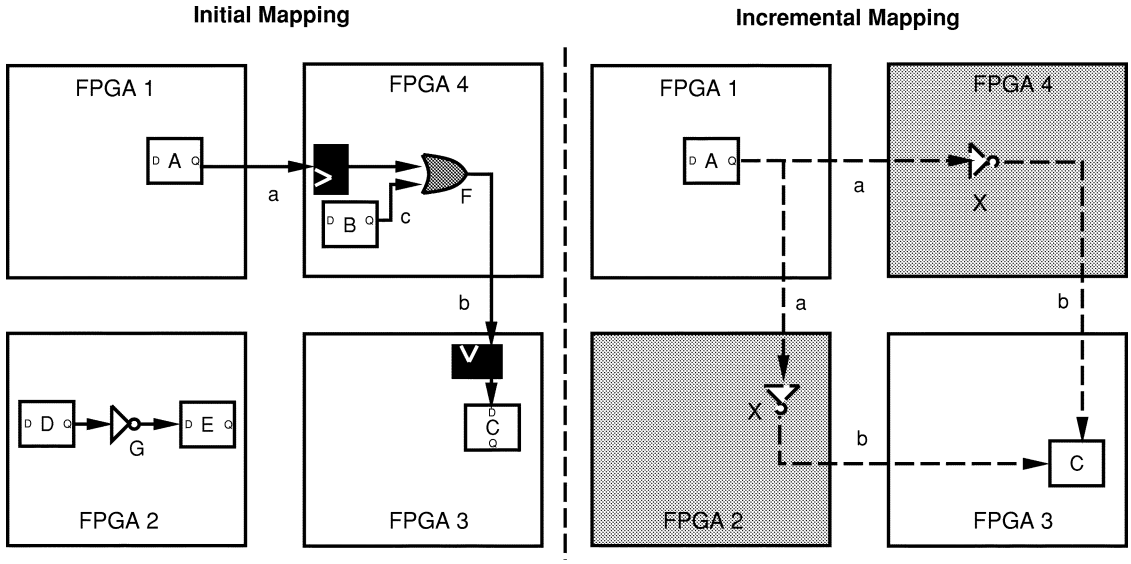


Fig. 11. An example illustrating the dependency of input and output links. The dashed lines in the mapping on the right illustrate potential paths for signals **a** and **b**.

piled to *transmit* signal $a$ to FPGA 2, and FPGA 3 will have to be recompiled to *receive* signal $b$ from FPGA 2.

For *interface* links that remain the same after design modification but interface to added logic, it is desirable to retain previously scheduled communication. Added design nodes that are driven by previously scheduled links should be *anchored* in an FPGA that is a sink for the link. As a result, the link will not have to be rescheduled and the source FPGA will not have to be recompiled. The corresponding argument can be made for modules that drive interface links.

The process of anchoring the affected logic can be performed in two stages: locating the affected nodes and assigning the nodes to FPGA partitions. Added logic that interfaces to unchanged links can be identified in a series of steps.

1) A depth-first search is performed on the modified design to determine if nets associated with the *original* link set $L$ remain present in the design.
2) For each net found to be present in the modified design, a check is made to determine if one of two conditions is met: 1) the net is driven by a node in the original design and fans out to an added node and 2) the net is driven by an added node and fans out to a node in the original design. These nets form a set $W$.
3) All nodes in $N_{\text{add}}$ attached to nets in $W$ form a set $P$.

After the affected set of nodes has been located, a placement heuristic is used to determine the appropriate destination FPGA for each node. The cost associated with placing a node $n$ in a specific FPGA $i$ is based on the number of interface links attached to the node $I_n$ and the number of these links routed to $i$ during initial design mapping $I$. Specifically, this cost is

$$\text{Cost}_f(i) = I_n - I. \tag{4}$$

After node cost has been determined for each FPGA, the node is assigned to the min-cost FPGA with sufficient capacity. Details of the node anchoring algorithm are shown in Fig. 12.

**W:** Set of nets from original design which drive/are driven by added nodes.
**P:** Set of nodes in $N_{add}$ connected to nets in $W$.

$Cost_f$**(i):** Cost of assigning a node to FPGA $i$

**For** each link in the original design.
  **If** link drives or is driven by a node in $N_{add}$.
    **Assign** associated net to $W$.
    **Add** node to $P$.
**EndFor**
**For** each node $n$ in $P$.
  **For** each FPGA $i$ in $B_{add}$.
    $Cost_f(i) = 0$.
    **For** each net in $W$ attached to node $n$ I/O.
      **If** net does not drive $i$ in initial mapping.
        $Cost_f(i)++$
    **EndFor**
  **EndFor**
  **Select** FPGA with minimum cost, $Cost_f(i)$.
  **Assign** node to FPGA with minimum cost, $Cost_f(i)$.
**EndFor**

Fig. 12. Algorithm for anchoring interface nodes to FPGAs.

*2) Partitioning Algorithm:* Incremental partitioning of added design logic onto modified FPGAs follows directly from the basic Kernighan–Lin, Fiduccia–Mattheyses (KLFM) [15] bipartitioning algorithm. To promote design quality, this algorithm has been supplemented with several optimizations to take unchanged logic and connections to the unchanged, fixed-placement FPGAs into account.

The KLFM partitioner distributes modules $N_{\text{add}}$ across available FPGAs $B_{\text{add}}$. Fig. 13, modified from [5], illustrates several partitioning steps in italics that reflect changes for incremental partitioning. These steps ensure that unchanged logic is reassigned to the same bin to which it was initially allocated by premodified design partitioning. Anchor nodes, evaluated prior to partitioning using the algorithm shown in Fig. 12, are assigned to selected FPGAs. During partitioning, anchor nodes can only

```
Assign unchanged nodes to previously assigned bins.
Add anchored nodes to represent interface links.
Randomly assign remaining new design nodes to
    balance bins.
While cutsize is reduced.
Unlock new design nodes.
    While valid moves exist
        Find unlocked node that most improves cutsize.
        Move whichever node most improves cutsize.
        Lock moved node.
        Update affected nets and node.
    endwhile
    Backtrack to point with minimum cutsize.
endwhile
```

Fig. 13. Modified KLFM algorithm.

shift to other partitions with the same anchor cost, as specified by (4). As a result of partitioning, each FPGA in $B_{\rm add}$ receives approximately the same amount of logic. Interconnect between the FPGAs in minimized so that routing can be completed with available inter-FPGA resources.

### D. Incremental Routing

Once modified logic has been distributed to FPGAs in $B_{\rm add}$, links in $L_{\rm add}$ are routed. Similar to initial-run routing, this process starts with depth and dependency analysis of the entire modified circuit using the approach described in Section II-C. Depth and dependency information for the modified design is used to supplement routing information from the initial design such as link send and arrival times $T_s$ and $T_a$.

*1) Routing Constraints:* An important aspect of depth analysis for modified circuits is the determination of which previously routed links from the initial design can remain in use. In some cases, portions of previously routed inter-FPGA links may need to be rerouted as a result of changed logic depth and dependency. As an example, consider the circuit shown in Fig. 14. The circuit is the same as that shown in Fig. 5 except that OR gate **F** and signals $e$ and $f$ have been added to the design. One potential incremental mapping for the modified circuit appears in Fig. 15.

A design clock cycle associated with the scheduled route of the circuit mapping in Fig. 15 is shown in Fig. 16. When these waveforms are compared to those in Fig. 7, it can be seen that an extra cycle of combinational delay has been added due to the OR gate evaluation in FPGA 3, extending the number of virtual cycles needed to evaluate the design. Closer examination of the two sets of waveforms indicates that although signal $b$ has previously been routed between FPGA 3 and FPGA 4 in the initial design, it will have to be rerouted for the modified mapping. For the initial design, signal $b$ has been routed between FPGA 3 and FPGA 4 on virtual cycle V4. As a result of the mapping shown in Fig. 15, signal $b$ cannot be routed until virtual cycle V5 due to combinational dependencies. This results in a need to recompile both FPGA 3 to transmit the signal on cycle V5 and FPGA 4 to receive the value on virtual cycle V5. In Section VI, it is shown that few combinational paths cover more than a few FPGAs, limiting the number of required fanout path ripups.

Although Fig. 15 shows OR gate $F$ assigned to FPGA 3, since both FPGA 2 and FPGA 3 are in $B_{\rm add}$, two possible targets existed during incremental partitioning. If the OR gate had been assigned to FPGA 2, the routing waveform for the modified design would have been the same as the waveform shown in Fig. 7 since all intra-FPGA combinational paths can be evaluated in one virtual cycle.

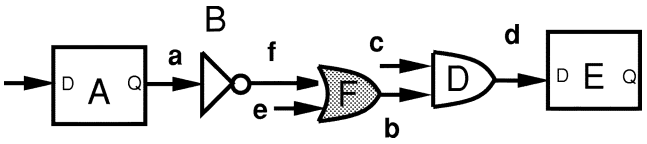


Fig. 14. A circuit modified from the example shown in Fig. 5. An OR gate **F** has been added to create this circuit.

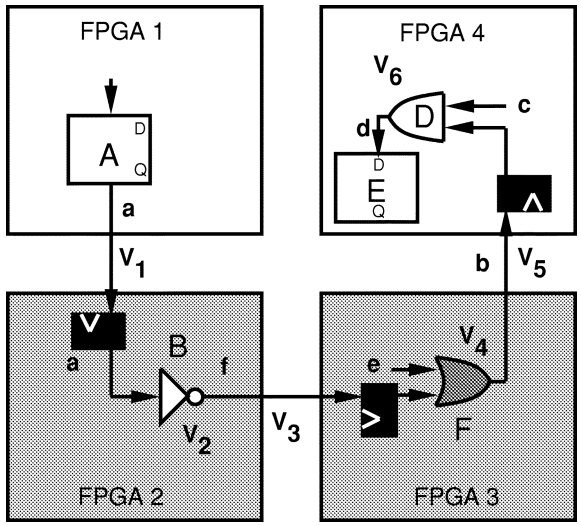


Fig. 15. An incremental mapping of the circuit in Fig. 14.

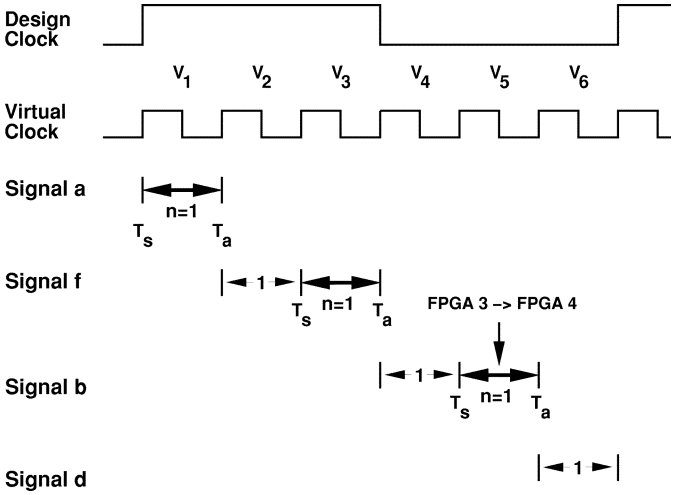


Fig. 16. Design clock cycle for the incremental mapping shown in Fig. 15.

*2) Routing Algorithm:* The goal of incremental scheduling is to minimize the number of virtual clock cycles needed to transport added links $L_{\rm add}$ between FPGAs in $B_{\rm add}$. The route scheduler assigns links to specific inter-FPGA channel wires on virtual clock cycles in which the resources are unused.

To support incremental routing, the channel graph from the original design mapping is used. Prior to routing, the graph is initialized to reflect the routing capacity consumed by the initial routes. Channel routing resources used by links in $L_{\rm sub}$ are

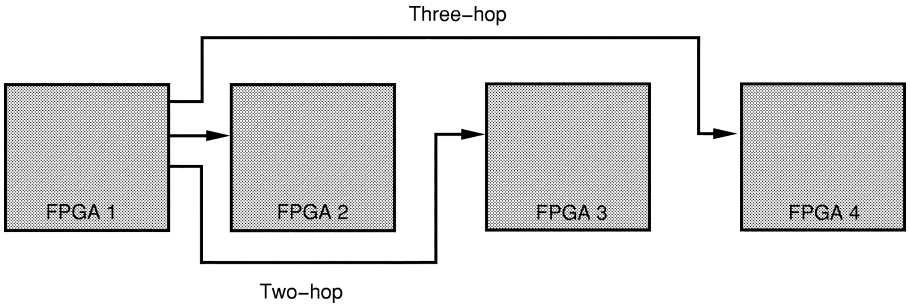


Fig. 17. FPGA topology connections.

not included in the initialization since they are not present in the modified design. A breadth-first analysis is performed on the entire modified design to identify fanout paths in unmodified logic that may be affected by depth or dependency changes, as described in Section IV-D1. The routing schedule is determined so that a minimum number of routes from initial mapping require rerouting and only FPGAs in $B_{\rm add}$ need to be recompiled.

Incremental routing involves both a check to determine if previous routes from the initial design mapping can still be used and the assignment of new $L_{\rm add}$ links to channel resources during specific time slots. The evaluation of previously routed links indicates whether initial virtual cycle designations can still be met in the potential presence of added combinational depth, as shown in Fig. 15. If previous send times $T_s$ cannot be met for a previously routed link, the routed link is ripped up and designated for rerouting. The evaluation of previously routed links takes place via dependency-checking for each link. Like initial-run routing, links are ordered for evaluation based on depth. Specific links are evaluated when the arrival times of all combinational inputs have been determined. This *ready time* $T_r$ [11] indicates the earliest point at which the link can be sent to neighboring FPGAs. If the link has been routed during initial-run routing and $T_r$ is less than the send time $T_s$ of the original route, the old link route and associated channel utilization can be preserved. Otherwise, the link is assigned to $L_{\rm add}$ for rerouting and previously used channel capacity is reclaimed.

After ordering via depth and dependency analysis, the specific steps needed to route links in $L_{\rm add}$ are the following.

1) The shortest path $P_{sd}$ from the source FPGA $s_f$ to the destination FPGA $d_f$ via FPGAs in $B_{\rm add}$ is determined. Note that through prior determination of $B_{\rm add}$, it is known that a feasible path exists.
2) The departure delay $T_d$ of the link is determined based on available channel capacity. Due to channel congestion, it may not be possible to route a link immediately at $T_r$. In the current route allocation approach, taken from [12], once a route leaves $s_f$, it is not delayed at any through-hop FPGAs. If both ready time and departure time are taken into account, the link leaves $s_f$ at the send time $T_s = T_r + T_d$.
3) As in initial-run routing, the arrival time $T_a$ at $d_f$ is $T_s+n$, where $n$ is the number of routing hops for the link.
4) After routing is complete, channel utilizations along the route path and ready times for dependent signals are updated.

In performing routes, the scheduler treats wires with more than one fanout as a set of two-terminal nets.

### E. Processor Compilation

The last step in the incremental compilation process involves the recompilation of affected FPGAs $B_{\rm add}$. In this work, recompilation of each individual FPGA was started from scratch; no information from initial-design FPGA place-and-route was used.

## V. Experimental Methodology

### A. The Effect of Interconnect Topology

To prove the practical benefits of the developed incremental compilation system, experimental results were obtained from mapping designs to both a commercial FPGA-based emulator and hypothetical emulators with varied inter-FPGA topologies. A set of experiments targeted an Ikos VirtuaLogic emulation system, which contains 64 Xilinx XC4036EX FPGAs. The FPGAs in this system are interconnected with both near-neighbor channels (14 unidirectional signals in each direction) between FPGAs and channels that connect to FPGAs two hops away (12 unidirectional signals in each direction to next-nearest neighbors). The developed compilation algorithms have been integrated with the VirtuaLogic compilation system, which performs initial design mapping. The result of both initial and incremental compilation is a set of Xilinx XC4036 bitstreams, which are programmed into the emulator.

Hypothetical emulation topologies were limited to inter-FPGA connectivity similar to the type shown in Fig. 17. Interconnect channels can be present between near-neighbor FPGAs and FPGAs separated by multiple hops. The effect of interconnect topology on system performance has previously been addressed in the context of FPGA-based logic emulation. Hauck evaluated bandwidth tradeoffs between near-neighbor connectivity and more distant connectivity [23]. It was found that if a topology contains too much near-neighbor interconnect, distant connections will require through-hops, potentially lengthening the delay of the critical routing path. If the topology contains insufficient near-neighbor bandwidth, local routes may be delayed due to congestion, extending the time needed to complete a user design cycle.

Increased connectivity between FPGAs provides a specific benefit for incremental compilation. In Section IV-A, it was noted that paths between FPGAs in $B_{\rm sub}$ may require the inclusion of *unmodified* FPGAs. If more direct FPGA-to-FPGA channels exist, fewer unmodified FPGAs may need to be recompiled, accelerating overall compile time. To evaluate the effect

TABLE II
BENCHMARK DESIGN STATISTICS

| Design | Source | Initial Gates | Modified Gates |
|---|---|---|---|
| DES | RAW | 124512 | 124581 |
| fir12 | RAW | 124585 | 124769 |
| ssp64 | RAW | 96577 | 96860 |
| spm16 | RAW | 127011 | 127992 |
| r4000 | Prep | 11826 | 12073 |

of topology on incremental compilation, we mapped both initial and modified designs to emulation systems with a variety of topologies consisting of near-neighbor, two-hop, and three-hop interconnect. Results that show the benefit of non-near-neighbor communication are presented in Section VI.

### B. Benchmark Designs

To evaluate the limitations of incremental compilation, several RTL benchmark circuits were first synthesized and then mapped to the VirtuaLogic emulation system. Following initial mapping, these benchmarks were modified at the RTL level and were mapped to the emulation system using the algorithms described in Section IV. Design statistics for the benchmarks are summarized in Table II. Sources for the benchmarks include the RAW [24] and Prep benchmark suites [25]. Subsequently, each benchmark is described along with a discussion of how the initial design was modified. Many of these modified benchmarks benefit from hardware specialization [26], the capability to directly customize a specific piece of hardware to perform a certain task. In several cases, the benchmark is specialized based on the value of constant parameters to reflect an optimization that can be integrated into the hardware.

DES is an established encryption standard that operates on 64-bit blocks of data and uses a 56-bit key [27]. Each 64-bit input value is encrypted to form a 64-bit block of output cypher text. Output data are the result of 16 stages of XOR operations involving intermediate results and the input key. Encryptions of multiple input values can be performed in parallel. To evaluate this design using logic emulation, an initial design was created based on a fixed 56-bit key and mapped to the logic emulator. The key value was then changed and a new, specialized hardware implementation was created. Although much of the DES circuit remains the same, important changes in structure occur in selected key input blocks (S-blocks) in the design.

Finite impulse response filters are integral parts of many digital signal-processing systems. Multipliers that scale sampled data by fixed constants are important components in many filter implementations. To evaluate incremental compilation, a 20-tap filter design requiring a fixed set of coefficients was synthesized and mapped to the VirtuaLogic emulator. A modified design was then created in RTL by changing the multiplier constant for two of the taps and resynthesizing the design. The modified design was then mapped to the emulator using incremental compilation.

Reconfigurable hardware has been used extensively to prototype hardware that can solve graph-based problems [28]. For these applications, graph functionality is embedded directly in hardware as a set of computational nodes and associated graph interconnect topology. For many graph-based applications, such as Boolean satisfiability, the embedded graph changes incrementally from application to application both in terms of node content and in terms of connectivity. To evaluate our system, initial RTL implementations of both a standard shortest path *(ssp64)* and multiplicative shortest path *(spm16)* graph were created using tools from the RAW benchmark suite [24]. After initial versions were synthesized and mapped to the emulator, versions with modified graph functionality were created and mapped to the emulator. To evaluate our system, both graph interconnect and node functionality modifications were made.

Microprocessor design often requires an evaluation of a number of different architectural choices. Part of this evaluation involves the determination of arithmetic logic unit (ALU) functionality. In an effort to evaluate the feasibility of performing incremental compilation on a modified RISC processor design, a new instruction was added to the instruction set of an existing, synthesizable RISC core. After initially mapping a standard R4000 design from the Prep benchmark suite [25] to the emulator, control logic and ALU support for an XNOR function were added to the design. This modified design was then synthesized and incrementally mapped to the emulator.

## VI. RESULTS

Both initial and modified versions of the benchmarks listed in Table II were mapped to the VirtuaLogic emulator. Results appear in Table III for both original and incremental mappings. Designs were initially partitioned onto the number of FPGAs shown in the *Compiled FPGAs* row in Table III. Following initial design mapping via the flow described in Section II-C, an analysis of results was performed to determine the verification behavior and emulation system utilization. FPGAs were compiled to run at a virtual clock speed of 34 MHz. Results listed in Table III indicate the critical and average path lengths in number of FPGAs, the number of virtual clocks per user design clock cycle, and the achieved emulation speed.

After successful initial implementation, design changes, as described in Section V, were made, and modified designs were partitioned across a minimum number of FPGAs $(B_{\mathrm{add}})$. As seen in the *Compiled FPGAs* row of Table III, fewer than 25% of FPGAs were recompiled for each design. For three of five designs, the run-time performance of the recompiled design matched the performance of the initial compilation. Extra virtual clock cycles for designs *ssp64* and *spm16* were a result of routing congestion along critical paths consisting of added design logic. The row marked *Rerouted links* indicates the number of previously routed links that were rerouted due to changes in dependencies, as described in Section IV-D. *Ave. VW I/O* indicates the average number of pins used on each FPGA, and *Ave. Hard I/O* indicates the average number of logic signals communicated for each partition.

The utilization of channel routing resources in a verification system varies over virtual clock cycles of the user design clock cycle. An example of channel utilization for design DES appears in Fig. 18. The solid line in the figure shows average bandwidth across all channels in the emulation system for the initial design, while the dashed line shows average bandwidth only in channels that are subsequently changed by incremental routing. The

TABLE III
RESULTS OF ORIGINAL AND INCREMENTAL COMPILATION FOR BENCHMARK DESIGNS

| | ssp64 | | spm16 | | DES | | fir12 | | r4000 | |
|---|---|---|---|---|---|---|---|---|---|---|
| | orig. | increm. | orig. | increm. | orig. | increm. | orig. | increm. | orig. | increm. |
| Total gates | 96577 | 96860 | 127011 | 127992 | 124512 | 124581 | 124585 | 124769 | 11826 | 12073 |
| Deleted gates | | 1825 | | 3218 | | 2534 | | 1815 | | 565 |
| Added gates | | 2108 | | 4199 | | 2603 | | 1999 | | 812 |
| Compiled FPGAs | 16 | 3 | 32 | 4 | 32 | 4 | 32 | 3 | 16 | 3 |
| Routed links | 2273 | 44 | 2277 | 67 | 6552 | 67 | 2133 | 68 | 1280 | 45 |
| Rerouted links | | 10 | | 8 | | 0 | | 5 | | 0 |
| Crit. path (FPGAs) | 5 | 5 | 5 | 5 | 4 | 4 | 14 | 14 | 14 | 12 |
| Ave. path (FPGAs) | 1.67 | 1.68 | 1.70 | 1.69 | 1.79 | 1.78 | 1.52 | 1.55 | 1.62 | 1.64 |
| Ave. VW I/O | 63 | 64 | 66 | 66 | 75 | 75 | 50 | 49 | 49 | 49 |
| Ave. Hard I/O | 473 | 478 | 242 | 243 | 410 | 405 | 133 | 130 | 160 | 152 |
| VW virtual clks | 15 | 17 | 12 | 14 | 12 | 12 | 21 | 21 | 19 | 19 |
| Emul. speed (MHz) | 2.27 | 2.00 | 2.83 | 2.43 | 2.83 | 2.83 | 1.62 | 1.62 | 1.79 | 1.79 |

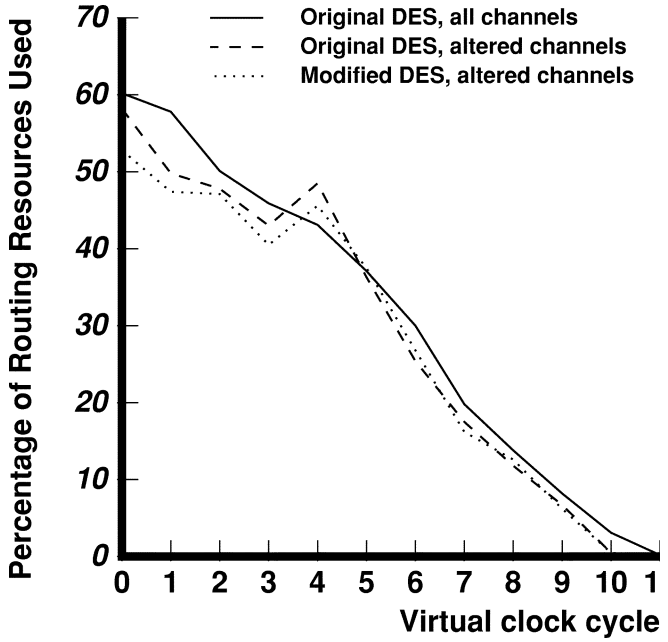


Fig. 18. Channel utilization per virtual clock cycle for a design clock cycle of DES. Note from Table III that the number of virtual clock cycles per design clock cycle for both original and modified DES designs is 12.

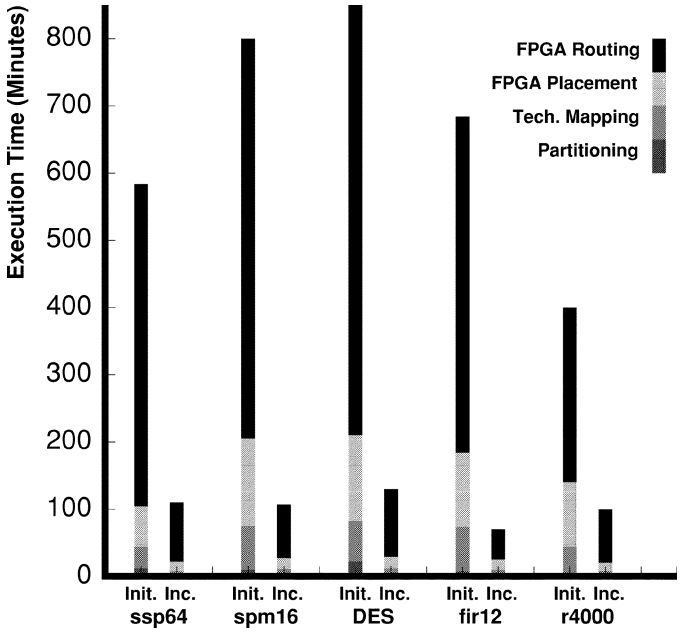


Fig. 19. Emulation system compilation time—initial and incremental.

dotted line indicates utilization in the same modified channels after incremental routing. As shown in Fig. 18, unused initial routing bandwidth can be used during incremental routing to complete link routes. In all cases, bandwidth requirements per virtual cycle are smaller as time passes due to signal dependencies. Due to static scheduling, the bandwidth per virtual clock cycle is the same for each user design clock cycle of a design.

Compile times for both initial and incremental design compilation appear in Fig. 19. All compilation was performed on a 360-MHz Sun SparcStation Ultra II with 512-MB RAM. For each design, incremental mapping time takes less than 20% of initial mapping time. In both cases, FPGA compilation takes the bulk of design mapping time. By limiting the number of affected FPGAs during system partitioning and routing, incremental compilation time can be kept to a minimum. Although the absolute compilation times are generally less for logic-processor-based versus FPGA-based parallel logic verification equipment, processor compilation time is still the limiting factor in overall compilation time.

After mapping initial and modified designs to an existing commercial logic emulator, the specified topology file used by the mapping software was modified to allow for mapping to emulation systems containing FPGAs with the same pin count but with varied inter-FPGA topologies. Both initial and incremental compilation were performed on each topology for designs *spm16* and *ssp64*. These experiments included topologies with near-neighbor interconnect and two-hop, three-hop, and four-hop interconnect in both horizontal and vertical dimensions. The number of virtual clock cycles per user design clock cycle for each design is shown in Table IV. The number of unidirectional channel wires allocated for each inter-FPGA distance is shown in the second row of the table. Channels in both horizontal and vertical dimensions were allocated identically. From the table, it can be seen that the topology used in the emulator (14 near-neighbor signals, 12 two-hop signals) provided both the best original and incremental design performance (minimal

TABLE IV
VIRTUAL CLOCK CYCLES REQUIRED PER DESIGN CLOCK CYCLE FOR VARIED TOPOLOGIES

| Design | [near] | [near, 2-hop] | [near, 2-hop, 3-hop] | [near, 2-hop, 3-hop, 4-hop] |
|---|---|---|---|---|
| | 26 wires | 14, 12 wires | 12, 10, 4 wires | 10, 8, 4, 4 wires |
| spm16-original | 18 | 12 | 24 | 24 |
| spm16-modified | 18 | 14 | 24 | 24 |
| ssp64-original | 18 | 15 | 24 | 24 |
| ssp64-modified | 19 | 17 | 24 | 24 |

virtual clock count). Topologies with only near-neighbor connections limit distant bandwidth to through-hop connections, extended verification time. Topologies that support connections between a larger collection of FPGAs limit available local bandwidth. This limitation leads to local route congestion and extended verification time.

## VII. CONCLUSIONS

In this paper, we have described and analyzed a set of compilation steps that can be used to map incrementally modified designs to parallel verification equipment. Important aspects of this paper include the development of constrained partitioning and routing algorithms optimized to map added design logic to verification system resources, such as FPGAs. After determining affected FPGA processors and evaluating paths between a minimum number of affected processors, a modified K-way partitioner distributes added logic to processors in the system. Subsequently, the interprocessor router creates connections between added logic such that original logic and routing are not affected. Finally, a minimum number of system processors are recompiled to form a working emulation model. To validate the benefit of this approach, five benchmark circuits have been mapped to a commercial FPGA-based Ikos VirtuaLogic emulator. For modest design changes (less than 15% of design logic), emulation system compile time has been reduced by at least a factor of five.

## REFERENCES


[1] *VirtuaLogic VLE-5 Emulation System Manual*, Ikos Systems, Inc., 2001.
[2] P. S. Tseng, "Reconfigured engines rev simulation," in *EE Times*, 2000.
[3] S. Walters, "Computer-aided prototyping for ASIC-based systems," *IEEE Design Test Comput.*, vol. 8, pp. 4–10, June 1991.
[4] (2002) System Explorer product brochure. Aptix Corporation. [Online]. Available: http://www.aptix.com/nova/literature/literature.html
[5] S. Hauck, "Multi-FPGA systems," Ph.D. dissertation, Dept. of Computer Science and Engineering, Univ. of Washington, 1995.
[6] H. Krupnova and G. Saucier, "FPGA-based emulation: Industrial and custom prototyping solutions," in *Proc. Field-Programmable Logic and Applications (FPL'2000)*, Villach, Austria, Aug. 2000, pp. 68–77.
[7] *Cobalt System Users Guide*, Quickturn Design Systems, Inc., 2001.
[8] L. Maliniak, "Hardware emulation draws speed from innovative 3D parallel processing based on custom IC's," *Electron. Design*, pp. 38–41, May 1994.
[9] (2001) Tharas Hammer product brief. Tharas Systems, Inc.. [Online]. Available: http://www.tharas.com
[10] J. Babb, R. Tessier, M. Dahl, S. Hanono, D. Hoki, and A. Agarwal, "Logic emulation with virtual wires," *IEEE Trans. Computer-Aided Design*, vol. 16, pp. 609–626, June 1997.
[11] S. Hauck and A. Agarwal, "Software technologies for reconfigurable computing," Department of Electrical and Computer Engineering, Northwestern Univ., 1996.
[12] C. Selvidge, A. Agarwal, M. Dahl, and J. Babb, "TIERS: Topology independent pipelined routing and scheduling for virtualwire compilation," in *Proc. ACM/SIGDA Int. Symp. Field-Programmable Gate Arrays*, Monterey, CA, Feb. 1995, pp. 25–31.
[13] H.-P. Su and Y.-L. Lin, "A phase assignment method for virtual-wire-based hardware emulation," *IEEE Trans. Computer-Aided Design*, vol. 16, pp. 776–783, July 1997.
[14] J. Cong and M. Sarrafzadeh, "Incremental physical design," in *Proc. ACM/IEEE Int. Symp. Physical Design*, 2000, pp. 84–92.
[15] C. M. Fiduccia and R. M. Mattheyses, "A linear-time heuristic for improved network partitions," in *Proc. Design Automation Conf.*, 1982, pp. 241–247.
[16] W. Shu and M.-Y. Wu, "Runtime incremental parallel scheduling for large-scale parallel computing," in *Proc. 5th Symp. Frontiers of Massively Parallel Computation*, Feb. 1995, pp. 456–463.
[17] G. DeMicheli, *Synthesis and Optimization of Digital Circuits*. New York: McGraw-Hill, 1994.
[18] R. Patasman, J. Lis, A. Nicolau, and D. Gajski, "Percolation based synthesis," in *Proc. 27th Design Automation Conf.*, June 1990, pp. 444–449.
[19] S. Hassoun, "Fine grain incremental rescheduling via architectural retiming," in *Proc. Int. Symp. System Synthesis*, Dec. 1998, pp. 158–163.
[20] R. Tessier, "Incremental compilation for logic emulation," in *Proc. 10th IEEE Int. Workshop Rapid Systems Prototyping*, Clearwater, FL, June 1999, pp. 236–241.
[21] X. Ma and E. L. Lloyd, "An incremental algorithm for broadcast scheduling in packet radio networks," in *Proc. IEEE Military Communications Conf.*, October 1998, pp. 205–210.
[22] A. Varma and S. Chalasani, "An incremental algorithm for TDM switching assignments in satellite and terrestrial networks," *IEEE J. Select. Areas Commun.*, vol. 10, pp. 364–377, Feb. 1992.
[23] S. Hauck, G. Borriello, and C. Ebeling, "Mesh routing topologies for FPGA arrays," *IEEE Trans. VLSI Syst.*, vol. 6, pp. 400–408, Sept. 1998.
[24] J. Babb, M. Frank, V. Lee, E. Waingold, and R. Barua, "The raw benchmark suite: Computation structures for general purpose computing," in *Proc. IEEE Symp. Field-Programmable Custom Computing Machines*, Napa, CA, Apr. 1997, pp. 161–171.
[25] *PREP Benchmark Suite #1, Version 1.3*: Programmable Electronic Performance Group, 1994.
[26] A. Dehon, "Reconfigurable architectures for general purpose computing," Ph.D. dissertation, Dept. of Electrical Engineering and Computer Science, Massachusetts Inst. of Technology, 1996.
[27] *Federal Information Processing Standards Publication*, 46-2, Dec. 1993.
[28] P. Zhong, M. Martonosi, P. Ashar, and S. Malik, "Accelerating Boolean satisfiability with configurable hardware," in *Proc. IEEE Symp. Field-Programmable Custom Computing Machines*, Napa, CA, Apr. 1998, pp. 186–195.



**Russell Tessier** (M'00) received the B.S. degree in computer engineering from Rensselaer Polytechnic Institute, Troy, NY, in 1989 and the S.M. and Ph.D. degrees in electrical engineering from the Massachusetts Institute of Technology, Cambridge, in 1992 and 1999, respectively.

He is an Assistant Professor of electrical and computer engineering at the University of Massachusetts (UMass), Amherst. He is a Founder of Virtual Machine Works, a logic emulation company, and has also worked at BBN and Ikos Systems. He currently leads the Reconfigurable Computing Group at UMass. His research interests include computer architecture, field-programmable gate arrays, and system verification.

**Snigdha Jana** received the B.E. degree in electronics from the University of Mumbai, India, in 1997 and the M.S. degree in electrical and computer engineering from the University of Massachusetts, Amherst, in 2000.

She has been with Intel Corporation, Hillsboro, OR, since September 2000. Currently, she is involved in the design and validation of next-generation microprocessors.